3
AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.

— David Wheeler

The process of generating WebAssembly binaries starts with the original source code, which is then processed by a compiler to produce a WebAssembly binary. This compiler is generally divided into three main components: a frontend that converts the source code into an intermediate representation, an optimizer/transformer that modifies this representation usually for performance, and a backend that compiles the final WebAssembly binary. This architecture is illustrated in the left most part of Figure 3.1.

Software Diversification, a preemptive security measure, can be integrated at various stages of this compilation process. However, applying diversification at the front-end has its limitations, as it would need a unique diversification mechanism for each language compatible with the frontend component. Conversely, diversification at later compiler stages, such as the optimizer or backend, offers a more practical alternative. This makes the latter stages of the compilers an ideal point for introducing practical Wasm diversification techniques. Our compiler-based strategies, represented in red and green in Figure 3.1, introduce a diversifier component into the optimizer/transformer and backend stages. This optimization/transformer component generates variants in the intermediate representation of a compiler, thereby creating artificial software diversity for WebAssembly. The variants are then compiled into WebAssembly binaries by the backend component of the compiler. Specifically, we propose two tools: CROW, which generates WebAssembly program variants, and MEWE, which packages these variants to enable multivariant execution [**?**]. Alternatively,

[0] Compilation probe time 2023/11/17 13:55:13

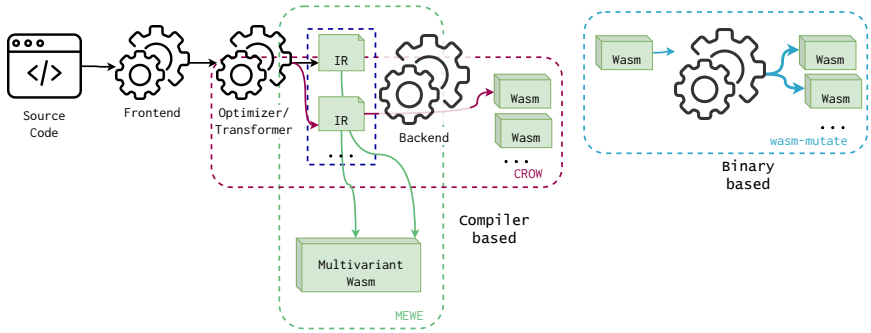

Figure 3.1: *Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.*

diversification can be directly applied to the WebAssembly binary, offering a language and compiler-agnostic approach. Our binary-based strategy, WASM-MUTATE, represented in blue in Figure 3.1, employs rewriting rules on an e-graph data structure to generate a variety of WebAssembly program variants.

This dissertation contributes to the field of Software Diversification for WebAssembly by presenting two primary strategies: compiler-based and binary-based. Within this chapter, we introduce three technical contributions: CROW, MEWE, and WASM-MUTATE. We also compare these contributions, highlighting their complementary nature. Additionally, we provide the artifacts for our contributions to promote open research and reproducibility of our main takeaways.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?**], represented as the red squared tooling in Figure 3.1. CROW is designed to produce functionally equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combination*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combination* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm

binaries.

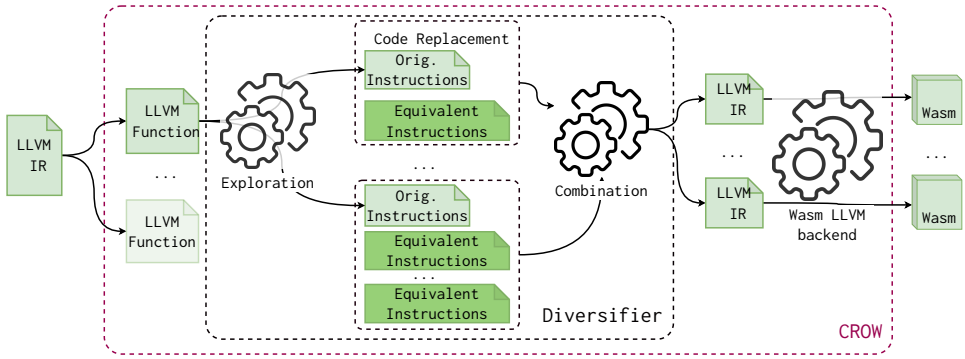

Figure 3.2: *CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [**?**].*

3.1.1 Enumerative synthesis

The cornerstone of CROW's exploration mechanism is its code replacement generation strategy, which is inspired by the superdiversifier methodology proposed by Jacob et al. [**?**]. The search space for generating variants is delineated through an enumerative synthesis process (see Enumerative synthesis in Section 2.2.1), which systematically produces all possible code replacements for each instruction in the original program. If a code replacement is identified to perform identically to the original program, it is reported as a functionally equivalent variant. This equivalence is confirmed using a theorem solver for rigorous verification.

Concretely, CROW is developed by extending the enumerative synthesis implementation found in Souper [**?**], an LLVM-based superoptimizer. Specifically, CROW constructs a Data Flow Graph for each LLVM instruction that returns an integer. Subsequently, it generates all viable expressions derived from a selected subset of the LLVM Intermediate Representation language for each DFG. The enumerative synthesis process incrementally generates code replacements, starting with the simplest expressions (those composed of a single instruction) and gradually increasing in complexity. The exploration process continues either until a timeout occurs or the size of the generated replacements exceeds a predefined threshold.

Notice that the search space increases exponentially with the size of the language used for enumerative synthesis. To mitigate this issue, we prevent CROW from synthesizing instructions without correspondence in the Wasm backend, effectively reducing the searching space. For example, creating an

expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end.

CROW is carefully designed to boost the generation of variants as much as possible. First, we disable the majority of the pruning strategies. Instead of preventing the generation of commutative operations during the searching, CROW still uses such transformation as a strategy to generate program variants. Second, CROW applies code transformations independently. For instance, if a suitable replacement is identified that can be applied at N different locations in the original program, CROW will generate 2^N distinct program variants, i.e., the power set of applying the transformation or not to each location. This approach leads to a combinatorial explosion in the number of available program variants, especially as the number of possible replacements increases.

Leveraging the ascending nature of its enumerative synthesis process, CROW is capable of creating variants that may outperform the original program in both size and efficiency. For instance, the first functionally equivalent transformation identified is typically the most optimal in terms of code size. This approach offers developers a range of performance options, allowing them to balance between diversification and performance without compromising the latter.

The last stage at CROW involves a custom Wasm LLVM backend, which generates the Wasm programs. For it, we remove all built-in optimizations in the LLVM backend that could reverse Wasm variants, i.e., we disable all optimizations in the Wasm backend that could reverse the CROW transformations.

3.1.2 Constant inferring

CROW inherently introduces a novel transformation strategy called *constant inferring*, which significantly expands the variety of WebAssembly program variants. Specifically, CROW identifies segments of code that can be simplified into a single constant assignment, with a particular focus on variables that control branching logic. After applying this *constant inferring* technique, the resulting program diverges substantially from the original program structure. This is crucial for diversification efforts, as one of the primary objectives is to create variants that are as distinct as possible from the original source code. In essence, the more divergent the variant, the more challenging it becomes to trace it back to its original form.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smallest number that fits with the Babbage condition in Line 4.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) %
            ↪ 1000000 != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n",
            ↪ current);
8       return 0 ;
9   }
```

Listing 3.1: *Babbage problem. Taken from [**?**].*

```
int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current)
        ↪ ;
    return 0 ;
}
```

Listing 3.2: *Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [**?**].*

CROW deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached [1]. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[2]

3.1.3 Exemplifying CROW

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

```
1  int f(int x) {
2      return 2 * x + x;
3  }
```

Listing 3.3: *C function that calculates the quantity $2x + x$.*

[1] In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large.

[2] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

```
define i32 @f(i32) {
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

  ret i32 %3
}
define i32 @main() {
  %1 = tail call i32 @f(
       i32 10)
  ret i32 %1
}
```

```
Replacement candidates
      for code_1
%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2
%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.4: *LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.*

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

Listing 3.5: *Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.*

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping.*

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 2
   i32.mul
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 1
   i32.shl
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   local.get 0
   i32.add
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 3
   i32.mul
```

Listing 3.6: *Wasm program variants generated from program Listing 3.3.*

Contribution paper and artifact

CROW is a compiler-based approach. It leverages enumerative synthesis to generate functionally equivalent code replacements and assembles them into diverse Wasm program variants. CROW uses SMT solvers to guarantee functional equivalence.
CROW is fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *at proceedings of Measurements, Attacks, and Defenses for the Web (MADWeb), NDSS 2021* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE) [**?**]. Execution path randomization is a technique that randomizes the execution path of a program at runtime, i.e. at each invocation of a function, a different variant is executed. MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

As illustrated in Figure 3.3, MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE gathers the LLVM IR variants created by CROW. The Mixer component, on the other hand, links the multivariant binary and creates a new entrypoint for the binary called *entrypoint tampering*. The tampering is needed in case the output of CROW are variants of the original entrypoint, e.g. the *main* function. Concretely, it wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint. The random generator is needed to perform the execution-path randomization. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. Finally, using the same custom Wasm LLVM backend as CROW, we generate a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

3.2.1 Multivariant call graph

The key component of MEWE consists of combining the variants into a single binary. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1 *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[***?** *]. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

3.2.2 Exemplifying a Multivariant binary

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

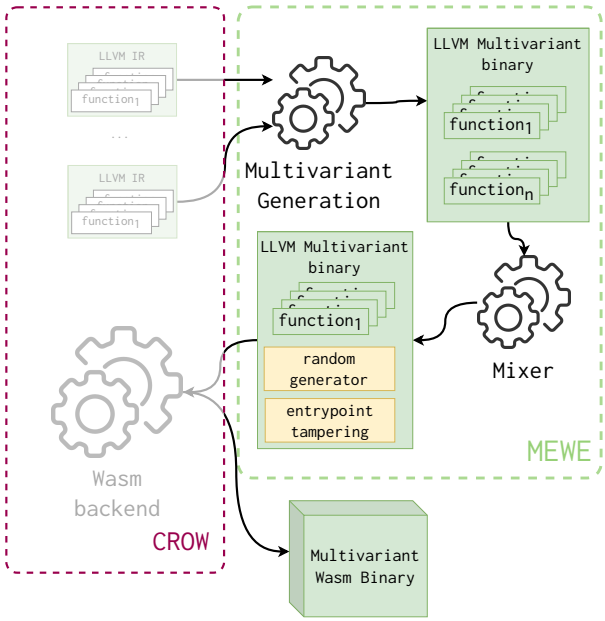

Figure 3.3: *Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure partially taken from [**?**].*

In Listing 3.7, we demonstrate how MEWE constructs the function dispatcher, corresponding to the rightmost green node in Figure 3.4, which handles three created variants including the original. The dispatcher function retains the same signature as the original function. Initially, the dispatcher invokes a random number generator, the output of which is used to select a specific function variant for execution (as seen on line 6 in Listing 3.7). To enhance security, we employ a switch-case structure within the dispatcher, mitigating vulnerabilities associated with speculative execution-based attacks [**?**] (refer to lines 12 to 19 in Listing 3.7). This approach also eliminates the need for multiple function definitions with identical signatures, thereby reducing the potential attack surface in cases where the function signature itself is vulnerable [**?**]. Additionally, MEWE can inline function variants directly into the dispatcher, obviating the need for redundant definitions (as illustrated on line 16 in Listing 3.7). Remarkably, we prioritize security over performance, i.e., while using indirect

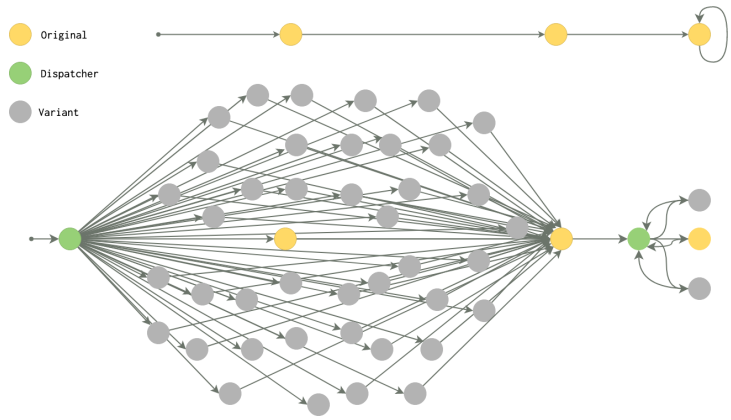

Figure 3.4: *Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [**?**].*

calls in place of a switch-case could offer constant-time performance benefits, we implement switch-case structures.

```
2  ; Multivariant foo wrapping ;
3  define internal i32 @foo(i32 %0) {
4      entry:
5        ; It first calls the dispatcher to discriminate between the created
             variants ;
6        %1 = call i32 @discriminate(i32 3)
7        switch i32 %1, label %end [
8          i32 0, label %case_43_
9          i32 1, label %case_44_
10       ]
11     ;One case for each generated variant of foo ;
12     case_43_:
13       %2 = call i32 @foo_43_(%0)
14       ret i32 %2
15     case_44_:
16       ; MEWE can inline the body of the a function variant ;
17       %3 = <body of foo_44_ inlined>
18       ret i32 %3
19     end:
20       ; The original is also included ;
21       %4 = call i32 @foo_original(%0)
22       ret i32 %4
23 }
```

Listing 3.7: *Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4. The code is commented for the sake of understanding.*

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. Notice that, the dispatcher function is constructed using the same signature as the original function. It first calls the random generator, which returns a value used to invoke a specific function variant (see line 6 in Listing 3.7). We utilize a switch-case structure in the dispatchers to prevent indirect calls, which are vulnerable to speculative execution-based attacks [**?**] (see lines 12 to 19 in Listing 3.7), i.e., the choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. In addition, MEWE can inline function variants inside the dispatcher instead of defining them again (see line 16 in Listing 3.7). Remarkably, we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

Contribution paper and artifact

MEWE provides dynamic execution path randomization by packaging variants generated out of CROW.
MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Proceedings of Moving Target Defense, 2022, ACM* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

3.3 WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [**?**], a tool that generates thousands of functionally equivalent variants out from a WebAssembly binary input. Leveraging rewriting rules and e-graphs [**?**] for software diversification, WASM-MUTATE synthesizes program variants by transforming parts of the original binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling.

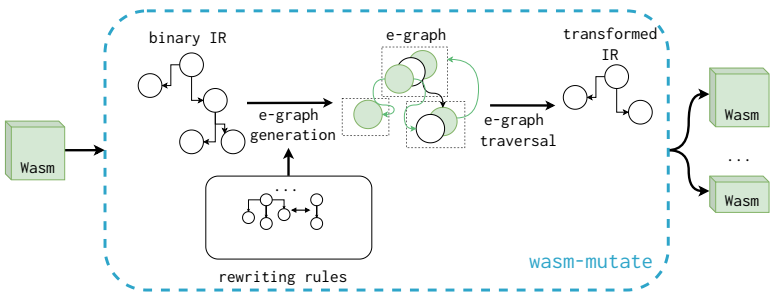

Figure 3.5: *WASM-MUTATE high-level architecture. It generates functionally equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.*

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the

rewriting rules. The assurance of functional equivalence is rooted in the inherent properties of the individual rewrite rules employed. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary.

WASM-MUTATE applies one transformation at a time. Notice that, the output of one applied transformation can be chained again as an input WebAssembly binary, enabling the generation of many variants, leading us to enunciate the notion of *Stacked transformation*

Definition 2 *Stacked transformation: Given an original input WebAssembly binary I and a diversifier D, stacked transformations are defined as the application of D over the binary I multiple times, i.e., $D(D(D(...(I))))$. Notice that, the number of stacked transformations are the number of times the diversifier D is applied.*

3.3.1 WebAssembly Rewriting Rules

WASM-MUTATE contains a comprehensive set of 135 rewriting rules. In this context, a rewriting rule is a tuple `(LHS, RHS, Cond)` where `LHS` specifies the segment of binary targeted for replacement, `RHS` describes its functionally equivalent substitute, and `Cond` outlines the conditions that must be met for the replacement to take place, e.g. enhancing type constraints. WASM-MUTATE groups these rewriting rules into meta-rules depending on their target inside a Wasm binary, ranging from high-level changes affecting binary section structure to low-level modifications within the code section. This section focuses on the biggest meta-rule implemented in WASM-MUTATE, the `Peephole` meta-rule [3].

Rewriting rules inside the *Peephole* meta-rule, operate over the data flow graph of instructions within a function body, representing the lowest level of rewriting. In WASM-MUTATE, we have implemented 125 rewriting rules specifically for this category, each one avoiding targeting instructions that might induce undefined behavior, e.g., function calls.

Moreover, we augment the internal representation of a Wasm program to bolster WASM-MUTATE's transformation capabilities through the `Peephole` meta-rule. Concretely, we augment the parsing stage in WASM-MUTATE by including custom operator instructions. These custom operator instructions are designed to use well-established code diversification techniques through rewriting rules. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program.

In the following example, we demonstrate a rewriting rule within the `Peephole` meta-rule that utilizes a custom `rand` operator to expand statically declared constants within any WebAssembly program function body. The unfolding

[3]For an in-depth explanation of the remaining meta-rules, refer to [**?**].

rewriting rule, as the name suggests, transforms statically declared constants into the sum of two random numbers. During the generation of the WebAssembly variant, the custom `rand` operator is substituted with a randomly chosen static constant. Notice that the condition specified in the last part of the rewriting rule ensures that this predicate is satisfied.

```
LHS i32.const x

---

RHS (i32.add (i32.rand i32.const y))

Cond y = x - i32.rand
```

Although this rewriting approach may appear simplistic, especially because compilers often eliminate it through *Constant Folding* optimization [**?**], it stresses on the spill/reload component of the compiler when the WebAssembly binary is JITed to machine code. Spill/reloads occur when the compiler runs out of physical registers to store intermediate calculations, resorting to specific memory locations for temporary storage. The unfolding rewriting rule indirectly stresses this segment of memory. Notably, with this specific rewriting rule, we have found a CVE in the wasmtime standalone engine [**?**].

3.3.2 E-Graphs traversals

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing and applying rewriting rules [**?**]. In the context of WASM-MUTATE, e-graphs are constructed from the input WebAssembly program and the implemented rewriting rules (we detail the e-graph construction process in Section 3 of [**?**]).

Willsey et al. highlight the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the functional equivalence of the derived code [**?**]. For instance, e-graphs solve the problem of providing the best code out of several optimization rules [**?**]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a functionally equivalent code variant.*

We exploit such property to fastly generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield functionally equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum

extraction depth of the expression, to prevent infinite recursion. Within the algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line 3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

Algorithm 1 e-graph traversal algorithm taken from [**?**].

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:      if depth = 0 then
3:          return smallest_tree_from(egraph, eclass)
4:      else
5:          nodes ← egraph[eclass]
6:          node ← random_choice(nodes)
7:          expr ← (node, operands = [])
8:          for each child ∈ node.children do
9:              subexpr ← TRAVERSE(egraph, child, depth − 1)
10:             expr.operands ← expr.operands ∪ {subexpr}
11:         return expr
```

3.3.3 Exemplifying WASM-MUTATE

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The former rule, grants the ability to append a `nop` instruction to any subexpression, a well-known low-level diversification strategy [**?**]. Conversely, the latter rule adds zero to any numeric value.

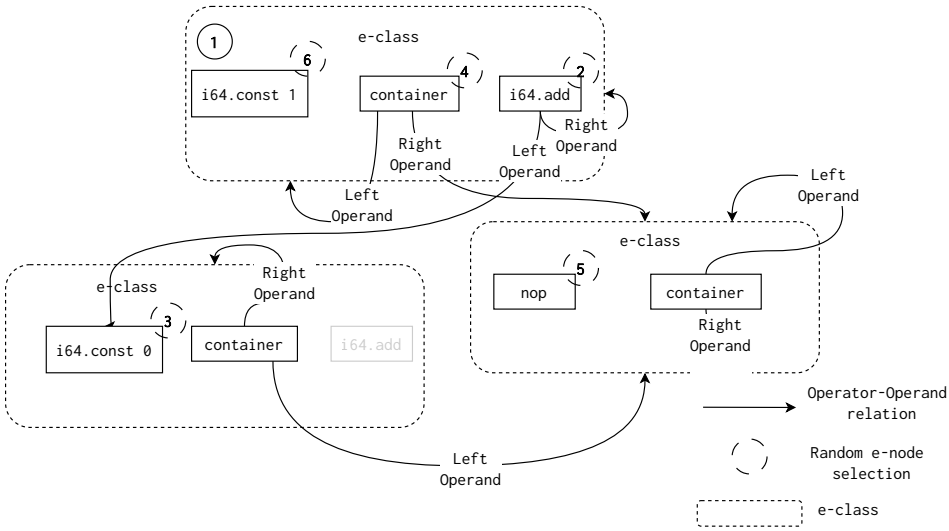

Figure 3.6: *e-graph built for rewriting the first instruction of Listing 3.8.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 3.8: *Wasm function.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: *Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.6. The textual format is folded for better understanding.*

Leveraging the code presented in Listing 3.8 alongside the defined rewriting rules, we build the e-graph, simplified in Figure 3.6. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At ②, it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr`

`= i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is choosen, selecting the `container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.6, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

Contribution paper and artifact

WASM-MUTATE uses hand-made rewriting rules and random traversals over e-graphs to provide a binary-based solution for WebAssembly diversification.
WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review at Computers & Security* `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the Bytecode Alliance organization [a]. The Bytecode Alliance is dedicated to creating secure new software foundations, building on standards such as WebAssembly and WASI.

[a] `https://bytecodealliance.org/`

3.4 Comparing CROW, MEWE, and WASM-MUTATE

TODO Mention that CROW can only affect peephole and deterministic code.

In this section, we compare CROW, MEWE, and WASM-MUTATE, highlighting their key differences. These distinctions are summarized in Table 3.1. The table is organized into columns that represent attributes of each tool: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary for diversification, strength of the generated variants, and the security applications of these variants. Each row in the table corresponds to a specific tool. The *Variant*

strength accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. For example, a higher value of the *Variant strength* indicates that the generated variants are not reversed by JIT compilers, ensuring that the diversification is preserved in an end-to-end scenario of a WebAssembly program, i.e. from the source code to its final execution. Notice that, the data and insights presented in the table are sourced from the respective papers of each tool and, from the previous discussion in this chapter.

CROW is a compiler-based strategy, needing access to the source code or its LLVM IR representation to work. Its core is an enumerative synthesis implementation with functional verification using SMT solvers, ensuring the functional equivalence of the generated variants. In addition, MEWE extends the capabilities of CROW, using its generated variants. It goes a step further by packaging the LLVM IR variants into a WebAssembly multivariant, providing MVE through execution path randomization. Both CROW and MEWE are fully automated, requiring no user intervention besides the input source code. WASM-MUTATE, on the other hand, is a binary-based tool. It uses a set of rewriting rules and the input Wasm binary to generate program variants, centralizing its core around random e-graph traversals. Remarkably, WASM-MUTATE removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary.

We have observed several interesting phenomena when aggregating the empirical data presented in the corresponding papers of CROW, MEWE and WASM-MUTATE [**? ? ?**]. This can be appreciated in the fourth, fifth and sixth columns of Table 3.1. We have observed that WASM-MUTATE generates more unique variants in one hour than CROW and MEWE in at least one order of magnitude. This is mainly because of three reasons. First, CROW and MEWE rely on SMT solvers to prove functionally equivalence, placing a bottleneck when generating variants. Second, CROW and MEWE generation capabilities are limited by the *overlapping* phenomenon discussed in Section 3.1.3. Third, WASM-MUTATE can generate variants in any part of the Wasm binary, while CROW and MEWE are limited to the code and function sections.

On the other hand, CROW and MEWE, by using enumerative synthesis, ensure that the generated variants are more preserved than the variants created by WASM-MUTATE. In other words, the transformations generated out of CROW and MEWE are virtually irreversible by JIT compilers, such as V8 and wasmtime. This phenomenon is highlighted in the *Variants strength* column of Table 3.1, where we show that CROW and MEWE generate variants with 96% of preservation against 75% of WASM-MUTATE. High preservation is especially important where the preservation of the diversification is crucial, e.g. to hinder reverse engineering.

Tool	**Input**	**Core**	**Variants in 1h**	**Target**	**Variants Strength**	**Security applications**
CROW	Source code or LLVM Ir	Enumerative synthesis with functional equivalence proved through SMT solvers	> 1k	Code section	**96%**	Hinders static analysis and reverse engineering.
MEWE	Source code or LL VM Ir	CROW, Multivariant execution	> 1k	Code and Function sections	**96%**	Hinders static and dynamic analysis, reverse engineering and, web timing-based attacks.
WASM-MUTATE	**Wasm binary**	hand-made rewriting rules, e-graph random traversals	**> 10k**	**All WebAssembly sections**	76%	Hinders signature-based identification, and cache timing side-channel attacks.

Table 3.1: *Comparing CROW, MEWE and WASM-MUTATE. The table columns are: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary, strength of the generated variants, and the security applications of these variants. The* Variant strength *accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. Our three technical contributions are complementary tools that can be combined.*

Takeaway

Our three technical contributions serve as complementary tools that can be combined. For instance, when the source code for a WebAssembly binary is either non-existent or inaccessible, WASM-MUTATE offers a viable solution for generating code variants. On the other hand, CROW and MEWE excel in scenarios where high preservation is crucial.

3.4.1 Security applications

The final column of Table 3.1 emphasizes the security benefits derived from the variants produced by our three key technical contributions. One immediate advantage of altering the structure of WebAssembly binaries across different variants is the mitigation of signature-based identification, thereby enhancing resistance to static reverse engineering. Additionally, our tools generate a diverse array of code variants that are highly preserved. This implies that these variants, each with their unique WebAssembly code, retain their distinct characteristics even after being translated into machine code by JIT compilers. This high level of preservation significantly mitigates the risks associated with side-channel attacks that target specific machine code instructions, such as port contention attacks [**?**]. For instance, if a WebAssembly binary is transformed in such a manner that its resulting machine code instructions differ from the original, it becomes more challenging for a side-channel attack. Conversely, if the compiler translates the variant into machine code that closely resembles the original, the side-channel attack could still exploit those instructions to extract information about the original WebAssembly binary.

Altering the layout of a WebAssembly program inherently influences its managed memory during runtime (see Section 2.1.3). This phenomenon is especially important for CROW and MEWE, given that they do not directly address the WebAssembly memory model. Significantly, CROW and MEWE considerably alter the managed memory by modifying the layout of the WebAssembly program. For example, the *constant inferring* transformations significantly alter the layout of program variants, affecting unmanaged memory elements such as the returning address of a function. Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform unmanaged memory instructions. Memory alterations, either to the unmanaged or managed memories, have substantial security implications, by eliminating potential cache timing side-channels [**?**].

Last but not least, our technical contributions enhance security against web timing-based attacks [**?**] by creating variants that exhibit a wide range of execution times, including faster variants compared to the original program.

This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks [**?**]. Adding another layer benefit from MEWE, the integration of diverse variants into multivariants can potentially disrupt dynamic reverse engineering tools such as symbolic executors [**?**]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

Takeaway

CROW, MEWE and WASM-MUTATE generate WebAssembly variants that can be used to enhance security. Overall, they generate variants that are suitable for hardening static and dynamic analysis, side-channel attacks, and, to thwart signature-based identification.

■ Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, proposing a novel e-graph traversal algorithm to fast spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, highlighting their respective benefits and limitations, alongside the potential security applications of the generated Wasm variants.

In Chapter 4, we present two use cases that support the exploitation of these tools. Chapter 4 serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

4 ASSESING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

If you find that you're spending all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.

— Donald Knuth

In this chapter, we illustrate the application of Software Diversification for both offensive and defensive purposes. We discuss two selected use cases that demonstrate practical applications of our contributions. Additionally, we discuss the challenges and benefits arising from the application of Software Diversification to WebAssembly.

4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [**?**]. This is due to the essence of cryptojacking, the faster the mining, the better. Let us illustrate how a malicious WebAssembly binary is involved into browser cryptojacking. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

[0] Compilation probe time 2023/11/17 13:55:13

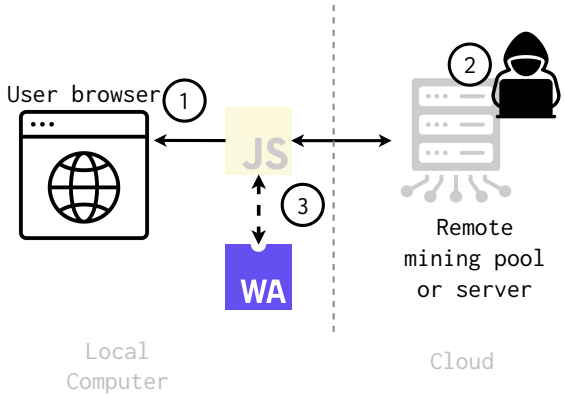

Figure 4.1: *A remote mining pool server, a JavaScript wrapper and the WebAssembly binary form the triad of a cryptojacking attack in browser clients.*

The aforementioned components require the following steps to succeed in cryptomining. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [**?**]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Section 2.1.5. However, malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [**?**], highlighting the critical importance of this use case. The case illustrated in the subsequent sections uses Offensive Software Diversification for evading malware detection in WebAssembly.

4.1.1 Cryptojacking defense evasion

Considering the previous scenario, several techniques can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. Such defense scenario is illustrated in Figure 4.2, where the WebAssembly malicious binary is blocked in ③. The primary aim of our use case is to investigate the effectiveness of code diversification as a means to circumvent cryptojacking defenses. Specifically, we assess whether the following evasion workflow can successfully bypass existing security measures:

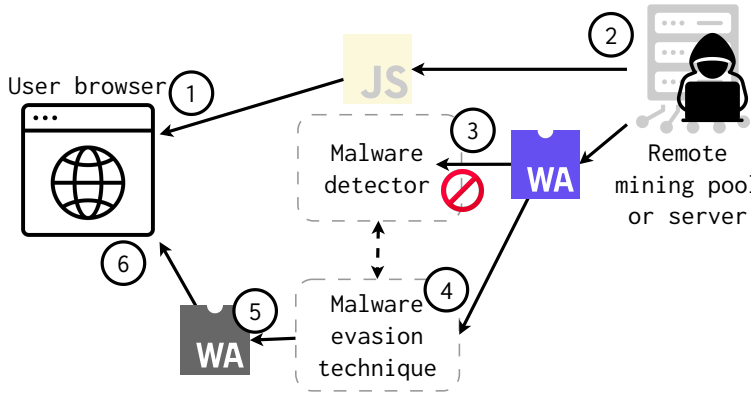

Figure 4.2: *Cryptojacking scenario in which the malware detection mechanism is bypassed by using an evasion technique.*

1. The user loads a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to ① and ② in Figure 4.2.

2. A malware detection mechanism (malware oracle) identifies and blocks malicious WebAssembly binaries at ③. For example, a network proxy could intercept and forward these resources to an external detection service via its API.

3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at ④.

4. The attacker delivers the modified binary instead of the original one ⑤, which initiates the cryptojacking process and compromises the browser ⑥. The detection method is not capable of detecting the malicious nature of the binary, and the attack is successful.

4.1.2 Methodology

Our aim is to empirically validate the workflow in Figure 4.2, i.e., using Offensive Software Diversification in evading malware detection systems. To achieve this, we employ WASM-MUTATE for generating WebAssembly malware variants. In this study, we categorize malware detection mechanisms as malware oracles, which can be of two types: binary and numeric. A binary oracle provides a binary decision, labeling a WebAssembly binary as either malicious or benign. In contrast, a numeric oracle returns a numerical value representing the confidence level of the detection.

Definition 3 *Malware oracle: A malware oracle is a detection mechanism that returns either a binary decision or a numerical value indicating the confidence level of the detection.*

We employ VirusTotal as a numeric oracle and MINOS [**?**] as a binary oracle. VirusTotal is an online service that analyzes files and returns a confidence score in the form of the number of antivirus that flag the input file as malware, thus qualifying as a numeric oracle. MINOS, on the other hand, converts WebAssembly binaries into grayscale images and employs a convolutional neural network for classification. It returns a binary decision, making it a binary oracle.

We use the wasmbench dataset [**?**] to establish a ground truth. After running the wasmbench dataset through VirusTotal and MINOS, we identify 33 binaries that are: 1) flagged as malicious by at least one VirusTotal vendor and, 2) are also detected by MINOS. Then, to simulate the evasion scenario in Figure 4.2, we use WASM-MUTATE to generate WebAssembly binary variants to evade malware detection (④ in Figure 4.2). We use WASM-MUTATE in two configurations: feedback-guided and stochastic diversification.

Definition 4 *Feedback-guided Diversification: In feedback-guided diversification, the transformation process of a WebAssembly program is guided by a numeric oracle, which influences the probability of each transformation. For instance, WASM-MUTATE can be configured to apply transformations that minimize the oracle's confidence score. Note that feedback-guided diversification needs a numeric oracle.*

Definition 5 *Stochastic Diversification: Unlike feedback-guided diversification, in stochastic diversification, each transformation has an equal likelihood of being applied to the input WebAssembly binary.*

Based on the two types of malware oracles and diversification configurations, we examine three scenarios: 1) VirusTotal with a feedback-guided diversification, 2) VirusTotal with an stochastic diversification, and 3) MINOS with a stochastic diversification. Notice that, the fourth scenario with MINOS and a feedback-guided diversification is not feasible, as MINOS is a binary oracle and cannot provide the numerical values required for feedback-guided diversification.

Our evaluation focuses on two key metrics: the success rate of evading detection mechanisms in VirusTotal and MINOS across the 33 flagged binaries, and the correctness of the generated variants.

Definition 6 *Evasion rate: This measures the efficacy of WASM-MUTATE in bypassing malware detection systems. For each flagged binary, we input it into WASM-MUTATE, configured with the selected oracle and diversification strategy. We then iteratively apply transformations to the output from the preceding step. This iterative process is halted either when the binary is no longer flagged by the oracle or when a maximum of 1000 stacked transformations have been applied (see Definition 2). This process is repeated with 10 random seeds per binary to*

simulate 10 different evasion experiments per binary.

Definition 7 *Correctness: This verifies the functional equivalence of the variants generated by WASM-MUTATE compared to the original binary. We execute the variants that entirely evade VirusTotal, using controlled and stochastic diversification configurations with WASM-MUTATE for both metrics. Our selection is limited to variants that allow us to fully reproduce the three components displayed in Figure 4.1. We then gather the hashes generated by the cryptojacking binaries and their generation speed, comparing these hashes with those from the original binary. If the hashes match, and the variant executes without error, with the minerpool component validating the hash, we can consider the variant as functionally equivalent.*

4.1.3 Results

In Table 4.1, we present a comprehensive summary of the evasion experiments presented in [**?**], focusing on two oracles: VirusTotal and MINOS[**?**]. The table is organized into two main categories to separate the results for each malware oracle. For VirusTotal, we further subdivide the results based on the two diversification configurations we employ: stochastic and feedback-guided diversification. In these subsections, the columns indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which the stochastic diversification or feedback-guided diversification setups best, the lower, the better. The MINOS section solely includes a column that specifies the number of transformations needed for complete evasion. The table has 33 + 1 rows, each representing a unique WebAssembly malware study subject. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.

Stochastic diversification to evade VirusTotal: We execute a stochastic diversification with WASM-MUTATE, setting a limit of 1000 iterations for each binary. In every iteration, we query VirusTotal to determine if the newly generated binary can elude detection. We repeat this procedure with ten distinct seeds for each binary, replicating ten different evasion experiments. As the stochastic diversification section of Table 4.1 illustrates, we successfully produce variants that fully evade detection for 30 out of 33 binaries. The average amount of iterations required to produce a variant that evades all detectors oscillates between 120 to 635 stacked transformations. The mean number of iterations needed never exceeds 1000 stacked transformations. However, three binaries remain detectable under the stochastic diversification setup. In these instances, the algorithm fails to evade 5 out of 31, 6 out of 30, and 5 out of 26 detectors. This shortfall can be attributed to the maximum number of iterations, 1000,

	VirusTotal					MINOS[?]
Hash	#D	Stochastic diversification		Feedback-guided diversification		
		Max. evaded	Mean trans.	Max. evaded	Mean trans.	Mean trans.
47d29959	31	**26**	N/A	19	N/A	100
9d30e7f0	30	**24**	N/A	17	N/A	419
8ebf4e44	26	**21**	N/A	13	N/A	92
c11d82d	20	20	**355**	20	446	115
0d996462	19	19	**401**	19	697	24
a32a6f4b	18	18	635	18	**625**	1
fbdd1efa	18	18	**310**	18	726	1
d2141ff2	9	9	**461**	9	781	81
aafff587	6	6	484	6	**331**	1
046dc081	6	6	404	6	**159**	33
643116ff	6	6	**144**	6	436	47
15b86a25	4	4	253	4	**131**	1
006b2fb6	4	4	**282**	4	380	1
942be4f7	4	4	200	4	200	29
7c36f462	4	4	236	4	**221**	85
fb15929f	4	4	**297**	4	475	1
24aae13a	4	4	**252**	4	401	980
000415b2	3	3	302	3	**34**	960
4cbdbbb1	3	3	295	3	**72**	1
65debcbe	2	2	131	2	**33**	38
59955b4c	2	2	130	2	**33**	38
89a3645c	2	2	431	2	**107**	108
a74a7cb8	2	2	124	2	**33**	38
119c53eb	2	2	104	2	**18**	1
089dd312	2	2	153	2	**123**	68
c1be4071	2	2	130	2	**33**	38
dceaf65b	2	2	140	2	132	66
6b8c7899	2	2	143	2	**33**	38
a27b45ef	2	2	145	2	**33**	33
68ca7c0e	2	2	137	2	**33**	38
f0b24409	2	2	127	2	**11**	33
5bc53343	2	2	118	2	**33**	33
e09c32c5	1	1	**120**	1	488	15
Median			218		131	38

Table 4.1: *The table has two main categories for each malware oracle, corresponding to the two oracles we use: VirusTotal and MINOS. For VirusTotal, divide the results based on the two diversification configurations: stochastic and feedback-guided diversification. We provide columns that indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which diversification setups are best, the lower, the better. The MINOS section includes a column that specifies the number of transformations needed for complete evasion. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.*

that we employ in our experiments. Increasing iterations further, however, seems unrealistic. If certain transformations enlarge the binary size, a significantly large binary could become impractical due to bandwidth limitations. In summary, stochastic diversification with WASM-MUTATE markedly reduces the detection rate by VirusTotal antivirus vendors for cryptojacking malware, achieving total evasion in 30 out of 33 (90%) cases within the malware dataset.

Feedback-guided diversification to evade VirusTotal: stochastic diversification does not guide the diversification based on the number of evaded detectors, it is purely random, and has some drawbacks. For example, some transformations might suppress other transformations previously applied. We have observed that, by carefully selecting the order and type of transformations applied, it is possible to evade detection systems in fewer iterations. This can be appreciated in the results of the feedback-guided diversification part of Table 4.1. The feedback-guided diversification setup successfully generates variants that totally evade the detection for 30 out of 33 binaries, it is thus as good as the stochastic setup. Remarkably, for 21 binaries out of 30, feedback-guided needs only 40% of the calls the stochastic diversification setup needs, demonstrating larger efficiency. Moreover, the lower number of transformations needed to evade detection, compared to the stochastic diversification setup, highlights the efficacy of the feedback-guided diversification setup in studying effective transformations. Consequently, malware detection system developers can leverage feedback-guided diversification to enhance their systems, focusing on identifying specific transformations.

Stochastic diversification to evade MINOS: Relying exclusively on VirusTotal for detection could pose issues, particularly given the existence of specialized solutions for WebAssembly, which differ from the general-purpose vendors within VirusTotal. In Section 2.1.5 we highlight several examples of such solutions. Yet, for its simplicity, we extend this experiment by using MINOS[**?**], an antivirus specifically designed for WebAssembly. The results of evading MINOS can be seen in the final column of Table 4.1. The bottom row of Table 4.1 highlights that fewer iterations are required to evade MINOS than VirusTotal through WebAssembly diversification, indicating a greater ease in eluding MINOS. The stochastic diversification setup requires a median iteration count of 218 to evade VirusTotal. In contrast, the feedback-guided diversification setup necessitates only 131 iterations. Remarkably, a mere 38 iterations are needed for MINOS. WASM-MUTATE evaded detection for 8 out of 33 binaries in a single iteration. This result implies the susceptibility of the MINOS model to binary diversification.

WebAssembly variants correctness: To evaluate the correctness of the malware variants created with WASM-MUTATE, we focused on six binaries that

we could build and execute end-to-end, as these had all three components outlined in Figure 4.1. We select only six binaries because the process of building and executing the binaries involves three components: the WebAssembly binary, its JavaScript complement, and the miner pool. These components were not found for the remaining 24 evaded binaries in the study subjects. For the six binaries, we then replace the original WebAssembly code with variants generated using VirusTotal as the malware oracle and WASM-MUTATE for both controlled and stochastic diversification configurations. We then execute both the original and the generated variants. We assess the correctness of the variants by examining the hashes they generate. Our findings show that all variants generated with WASM-MUTATE are correct, i.e., they generate the correct hashes and execute without error. Additionally, we found that 19% of the generated variants surpassed the original cryptojacking binaries in performance.

Reflection

Malware detection presents a challenging and well known issue [**?**]. However, previous research on static WebAssembly malware detection do not consider obfuscation techniques to augment their findings. Specifically, the current literature acknowledges only metadata (WebAssembly custom sections) obfuscation or the total absence of obfuscation techniques for WebAssembly [**?** **?** **?** **?** **?**]. As explored in Section 2.2, a software diversification engine could potentially serve as an obfuscator. We exhibit this potential with WASM-MUTATE. As a result, our software diversification tools offer a feasible method to improve the precision of WebAssembly malware detection systems. Existing tools could enhance their evaluation dataset of WebAssembly malwares by incorporating the variants generated by WASM-MUTATE.

Contribution paper

WASM-MUTATE generates correct and performant variants of WebAssembly cryptojacking that successfully evade malware detection. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" *at Computers & Security, 2023* `https://www.sciencedirect.com/science/article/pii/S0167404823002067`.

4.2 Defensive Diversification: Speculative Side-channel protection

As discussed in Section 2.1, WebAssembly is quickly becoming a cornerstone technology in backend systems. Leading companies like Cloudflare and Fastly are championing the integration of WebAssembly into their edge computing platforms, thereby enabling developers to deploy applications that are both modular and securely sandboxed. These server-side WebAssembly applications are generally architected as isolated, single-responsibility services, a model referred to as Function-as-a-Service (FaaS) [? ?]. The operational flow of WebAssembly binaries in FaaS platforms is illustrated in Figure 4.3.

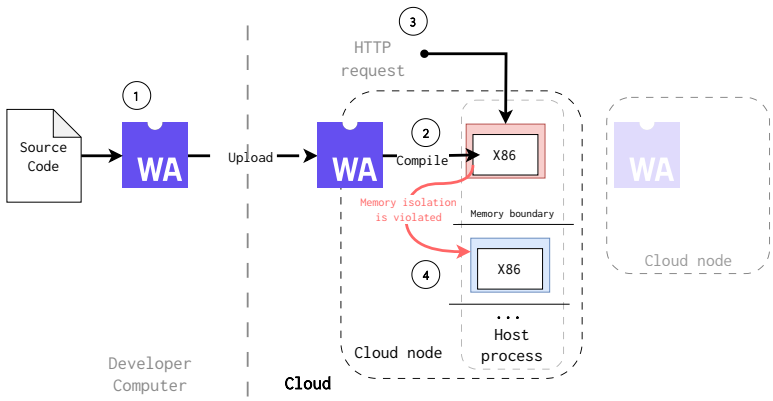

Figure 4.3: *WebAssembly binaries on FaaS platforms. Developers can submit any WebAssembly binary to the platform to be executed as a service in a sandboxed and isolated manner. Yet, WebAssembly binaries are not immune to Spectre attacks.*

The fundamental advantage of using WebAssembly in FaaS platforms lies in its ability to encapsulate thousands of WebAssembly binaries within a singular host process. A developer could compile its source code into a WebAssembly program suitable for the cloud platform and then submit it (① in Figure 4.3). This host process is then disseminated across a network of servers and data centers (② in Figure 4.3). These platforms convert WebAssembly programs into native code, which is subsequently executed in a sandboxed environment. Host processes can then instantiate new WebAssembly sandboxes for each client function, executing them in response to specific user requests with nanosecond-level latency (③ in Figure 4.3). This architecture inherently isolates WebAssembly binary executions from each other as well as from the host process, enhancing security.

However, while WebAssembly is engineered with a strong focus on security and isolation, it is not entirely immune to vulnerabilities such as Spectre attacks [? ?] (④ in Figure 4.3). In the sections that follow, we explore how

software diversification techniques can be employed to harden WebAssembly binaries against such attacks.

4.2.1 Threat model: speculative side-channel attacks

To illustrate the threat model concerning WebAssembly programs in FaaS platforms, consider the following scenario. Developers, including potentially malicious actors, have the ability to submit any WebAssembly binary to the FaaS platform. A malicious actor could then upload a WebAssembly binary that, once compiled to native code, employs Spectre attacks. Spectre attacks exploit hardware-based prediction mechanisms to trigger mispredictions, leading to the speculative execution of specific instruction sequences that are not part of the original, sequential execution flow. By taking advantage of this speculative execution, an attacker can potentially access sensitive information stored in the memory allocated to other WebAssembly instance (including itself by violating Control Flow Integrity) or even the host process. Therefore, this poses a significant risk for the overall execution system.

Narayan and colleagues [**?**] have categorized potential Spectre attacks on WebAssembly binaries into three distinct types, each corresponding to a specific hardware predictor being exploited and a particular FaaS scenario: Branch Target Buffer Attacks, Return Stack Buffer Attacks, and Pattern History Table Attacks defined as follows:

1. The Spectre Branch Target Buffer (btb) attack exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target.

2. The Spectre Return Stack Buffer (rsb) attack exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions.

3. The Spectre Pattern History Table (pht) takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition.

4.2.2 Methodology

Our goal is to empirically validate that Software Diversification can effectively mitigate the risks associated with Spectre attacks in WebAssembly binaries. The green-highlighted section in Figure 4.4 illustrates how Software Diversification can be integrated into the FaaS platform workflow. The core idea is to generate unique and diverse WebAssembly variants that can be randomized at the time of deployment. For this use case, we employ WASM-MUTATE as our tool for Software Diversification.

Program	Attack
btb_breakout	Spectre branch target buffer (btb)
btb_leakage	Spectre branch target buffer (btb)
ret2spec	Spectre Return Stack Buffer (rsb)
pht	Spectre Pattern History Table (pht)

Table 4.2: *WebAssembly program name and its respective attack.*

To empirically demonstrate that Software Diversification can indeed mitigate Spectre vulnerabilities, we reuse the WebAssembly attack scenarios proposed by Narayan and colleagues in their work on Swivel [**?**]. Swivel is a compiler-based strategy designed to counteract Spectre attacks on WebAssembly binaries by linearizing their control flow during machine code compilation. Our approach differs from theirs in that it is binary-based, compiler-agnostic, and platform-agnostic; we do not propose altering the deployment or toolchain of FaaS platforms.

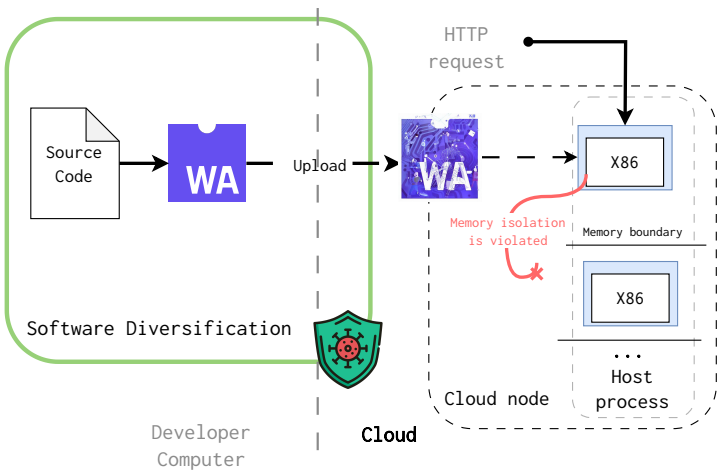

Figure 4.4: *Diversifying WebAssembly binaries to mitigate Spectre attacks in FaaS platforms.*

To measure the efficacy of WASM-MUTATE in mitigating Spectre, we diversify four WebAssembly binaries proposed in the Swivel study. The names of these programs and the specific attacks we examine are available in Table 4.2. For each of these four binaries, we generate up to 1000 random stacked transformations (see Definition 2) using 100 distinct seeds, resulting in a total of 100,000 variants for each original binary. At every 100th stacked transformation for each binary and seed, we assess the impact of diversification on the Spectre

attacks by measuring the attack bandwidth for data exfiltration.

Definition 8 *Attack bandwidth: Given data* $D = \{b_0, b1, ..., b_C\}$ *being exfiltrated in time* T *and* $K = k_0, k_1, ..., k_N$ *the collection of correct data bytes, the bandwidth metric is defined as:*

$$\frac{|b_i \text{ such that } b_i \in K|}{T}$$

The previous metric not only captures the success or failure of the attacks but also quantifies the extent to which data exfiltration is hindered. For example, a variant that still leaks data but does so at an impractically slow rate would be considered hardened against the attack.

4.2.3 Results

Figure 4.5 offers a graphical representation of WASM-MUTATE's influence on the Swivel original programs: btb_breakout and btb_leakage with the btb attack. The Y-axis represents the exfiltration bandwidth (see Definition 8). The bandwidth of the original binary under attack is marked as a blue dashed horizontal line. In each plot, the variants are grouped in clusters of 100 stacked transformations. These are indicated by the green violinplots.

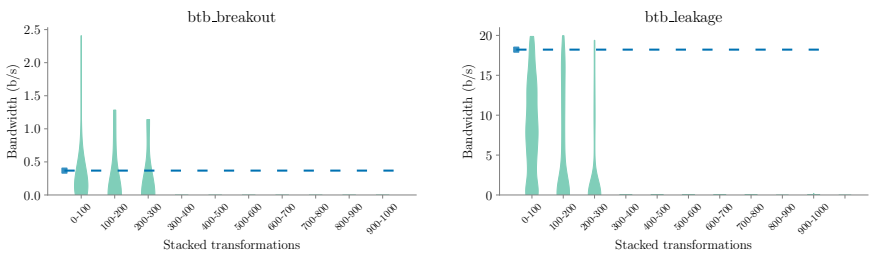

Figure 4.5: *Impact of WASM-MUTATE over btb_breakout and btb_leakage binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for all 100000 variants generated out of each original program, 70% have less data leakage bandwidth. After 200 stacked transformations, the exfiltration bandwidth drops to zero.*

Population Strength: For the binaries btb_breakout and btb_leakage, WASM-MUTATE exhibits a high level of effectiveness, generating variants that leak less information than the original in 78% and 70% of instances, respectively. For both programs, after applying 200 stacked transformations, the exfiltration bandwidth drops to zero. This implies that WASM-MUTATE is capable of synthesizing variants that are entirely protected from the original attack. If

we consider the results in Table 3.1, generating a variant with 200 stacked transformations can be accomplished in just a matter of seconds for a single WebAssembly binary.

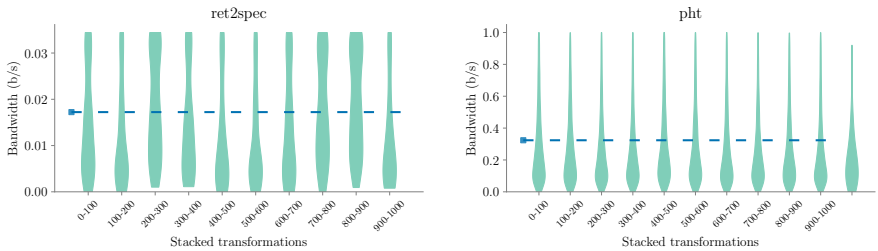

Figure 4.6: *Impact of WASM-MUTATE over ret2spec and pht binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for both programs approximately 70% of the variants have less data leakage bandwidth.*

Effectiveness of WASM-MUTATE: As illustrated in Figure 4.6, similarly to Figure 4.5, WASM-MUTATE significantly impacts the programs ret2spec and pht when subjected to their respective attacks. In 76% of instances for ret2spec and 71% for pht, the generated variants demonstrated reduced attack bandwidth compared to the original binaries. The plots reveal that a notable decrease in exfiltration bandwidth occurs after applying at least 100 stacked transformations. While both programs show signs of hardening through reduced attack bandwidth, this effect is not immediate and requires a substantial number of transformations to become effective. Additionally, the bandwidth distribution is more varied for these two programs compared to the two previous ones. Our analysis suggests a correlation between the reduction in attack bandwidth and the complexity of the binary being diversified. Specifically, ret2spec and pht are substantially larger programs, containing over 300,000 instructions, compared to btb_breakout and btb_leakage, which have fewer than 800 instructions. Therefore, given that WASM-MUTATE performs one transformation per invocation, the probability of affecting critical components to hinder attacks decreases in larger binaries.

Disrupting timers: Cache timing side-channel attacks, including for the four binaries analyzed in this use case, depend on precise timers to measure cache access times. Disrupting these timers can effectively neutralize the attack [**?**]. One key reason our results show variants resilient to Spectre attacks is the approach of WASM-MUTATE. It creates variants that offer a similar approach. Our WebAssembly variants introduce perturbations in the timing steps of WebAssembly variants. This is illustrated in Listing 4.1 and Listing 4.2, where

the former shows the original time measurement and the latter presents a variant with introduced operations. By introducing additional instructions, the inherent randomness in the time measurement of a single or a few instructions is amplified, thereby reducing the timer's accuracy.

```
;; Code from original btb_breakout
...
(call $readTimer)
(set_local $end_time)
... access to mem
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 4.1: *Wasm timer code.*

```
;; Variant code
...
(call $readTimer)
(set_local $end_time)
<inserted instructions>
... access to mem
<inserted instructions>
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 4.2: *WebAssembly variant with more instructions added in between time measurement.*

Padding speculated instructions: CPUs have a limit on the number of instructions they can cache. WASM-MUTATE injects instructions to exceed this limit. This effectively disables the speculative execution of memory accesses. This approach is akin to padding [**?**], as demonstrated in Listing 4.3 and Listing 4.4. This padding disrupts the binary code's layout in memory, hindering the attacker's ability to initiate speculative execution. Even if speculative execution occurs, the memory access does not proceed as the attacker intended.

```
;; Code from original btb_breakout
...
;; train the code to jump here (index 1)
(i32.load (i32.const 2000))
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
(i32.load (i32.const 339968)) ;; S(83) is the secret
(i32.store (i32.const 83)) ;; just prevent optimization
```

Listing 4.3: *Two jump locations. The top one trains the branch predictor, the bottom one is the expected jump that exfiltrates the memory access.*

```
;; Variant code
...
;; train the code to jump here (index 1)
<inserted instructions>
(i32.load (i32.const 2000))
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
<inserted instructions>
(i32.load (i32.const 339968)) ;; "S"(83) is the secret
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
```

Listing 4.4: *WebAssembly variant with more instructions added indindinctly between jump places.*

Managed memory impact: The success in diminishing Spectre attacks is mainly explained by the fact that WASM-MUTATE synthesizes variants that effectively alter memory access patterns. We have identified four primary factors responsible for the divergence in memory accesses among WASM-MUTATE generated variants. First, modifications to the binary layout—even those that do not affect executed code—inevitably alter memory accesses within the program's stack. Specifically, WASM-MUTATE generates variants that modify the return addresses of functions, which consequently leads to differences in execution flow and memory accesses. Second, one of our rewriting rules incorporates artificial global values into WebAssembly binaries. The access to these global variables inevitably affects the managed memory (see Section 2.1.3). Third, WASM-MUTATE injects 'phantom' instructions which do not aim to modify the outcome of a transformed function during execution. These intermediate calculations trigger the spill/reload component of the wasmtime compiler, varying spill and reload operations. In the context of limited physical resources, these operations temporarily store values in memory for later retrieval and use, thus creating

diverse managed memory accesses (see the example at Section 3.3.1). Finally, certain rewriting rules implemented by WASM-MUTATE replicate fragments of code, e.g., performing commutative operations. These code segments may contain memory accesses, and while neither the memory addresses nor their values change, the frequency of these operations does.

Reflection

Beyond Spectre, one can use WASM-MUTATE to mitigate other side-channel attacks. For instance, port contention attacks [**?**] rely on the execution of specific instructions for a successful attack. Not only WASM-MUTATE, but also our other tools, can alter those instructions, thereby mitigating the attack. The effectiveness of WASM-MUTATE, coupled with its ability to generate numerous variants, establishes it as an apt tool for mitigating side-channel attacks. Consider, for example, applying this on a global FaaS platform scale. In this scenario, one could deploy a unique, hardened variant for each machine and even for every fresh WebAssembly spawned per user request.

Contribution paper

WASM-MUTATE crafts WebAssembly binaries that are resilient to Spectre-like attacks. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* `https://arxiv.org/pdf/2309.07638.pdf`.

■ Conclusions

In this chapter, we explore Offensive and Defensive Software Diversification applied to WebAssembly. Offensive Software Diversification highlights both the potential and the latent security risks in applying Software Diversification to WebAssembly malware. Our findings suggest potential enhancements to the automatic detection of cryptojacking malware in WebAssembly, e.g., by stressing their resilience with WebAssembly malware variants. Conversely, Defensive Software Diversification serves as a proactive guard, specifically designed to mitigate the risks associated with Spectre attacks.

Moreover, we have conducted experiments with various use cases that are not shown in this chapter. For instance, CROW [**?**] excels in generating WebAssembly variants that minimize side-channel noise, thereby bolstering defenses against potential side-channel attacks. Alternatively, deploying multivariants from MEWE [**?**] can thwart high-level timing-based side-channels

[**?**]. Specifically, we conducted experiments on the round-trip times of the generated multivariants and concluded that, at a high level, the timing side-channel information cannot discriminate between variants. In the subsequent chapter, we will summarize the primary conclusions of this dissertation and propose avenues for future research.